

# TI Übung 6

## Koordinierung

Andreas I. Schmied (schmied@inf...)

Abteilung Verteilte Systeme  
Universität Ulm

SS2005

# Und nun...

## 1 Wiederholung

# Wiederholung – Gegenseitiger Ausschluss

- mit Verfahren von Peterson
- mit binärem Semaphor (Mutex)
  - `class Semaphor int s=1; ...`
  - Methoden `p`, `v` atomar
  - `Semaphor.p( )`
    - `s--` falls `s>0`
  - `Semaphor.v( )`
    - `s++`
- und weitere Verfahren je nach Einsatzgebiet...

# Wiederholung – kleine Vorschau

- PV-Chunk Semaphor
  - verändert ggf. um mehr als Eins
  - immer noch atomar!
  - P(int amount)
    - blockiert falls  $s < \text{amount}$
  - V(int amount)
- belegen/freigeben mehrerer Ressourcen
- siehe Vorlesung: Leser/Schreiber-Probleme

# Wiederholung – Falsche Koordinierung

- Deadlock
  - stabile Verklemmung
  - zyklische Blockierung
- Lifelock
  - u.U. schwer erkennbar
  - Arbeiten ohne Fortschritt
  - z.B. zyklisches aktives Warten
- unbewusst falsche „P-V“-Reihung
  - Deadlock bei mehreren P („Fail-Stop“)
  - keine Koordinierung bei mehreren V oder V ohne vorherigem P
    - Katastrophe, u.U. schwer erkennbar
- falsche Sequenz mehrerer Semaphoren
  - siehe Mutex vs. Empty/Full bei „Bounded Buffer“

# Wiederholung – Threading mit Java

- Interface Runnable
  - für „aktive Objekte“
  - Methode void run();
- Klasse Thread
  - implementiert Runnable
  - Methode void start(); startet neuen Thread
  - **keinesfalls run() direkt ausführen** → keine Nebenläufigkeit!
- Beispiel

```
1 class Worker extends Thread {  
2     public void run() { .. do something .. }  
3 }  
4 ...  
5 Thread t = new Worker();  
6 t.start();
```

# Und nun...

## 2 Aufgabe: Mutex

# Aufgabe: Mutex – Aufgabe

- Skizzieren Sie ein Programm...
  - das beliebig oft nebenläufig ausgeführt werden kann
  - in einer Endlosschleife
  - zuerst einen kritischen Abschnitt
  - dann einen unkritischen abarbeitet
- Nutzen Sie dazu folgende **lückenhafte** Test-And-Set-Implementierung
  - vervollständigen Sie zuerst den Bit-Quellcode
  - test/clear sollen atomar ausgeführt werden
  - alle Programmthreads sollen auf dasselbe Bit zugreifen
- welche Nachteile sehen Sie an dieser Vorgehensweise?
  - welche Auswege gäbe es?

# Aufgabe: Mutex – Bit-Impl. (unvollständig)

```
1 public class Bit {  
2  
3     boolean flag=false;  
4  
5     public boolean test() {  
6         boolean old = flag;  
7         flag=true;  
8         return old;  
9     }  
10  
11     public void clear() {  
12         flag=false;  
13     }  
14 }
```

# Aufgabe: Mutex – Bit-Impl. (vollständig)

```
1 public class Bit {  
2  
3     private boolean flag=false;  
4  
5     synchronized public boolean test() {  
6         boolean old = flag;  
7         flag=true;  
8         return old;  
9     }  
10  
11     synchronized public void clear() {  
12         flag=false;  
13     }  
14 }
```

# Aufgabe: Mutex – Lösungsskizze

```
1 public class Worker extends Runnable {
2
3     static Bit bit = new Bit();
4
5     public void run() {
6         while(true) {
7             while(bit.test());
8             ... k.A. ...
9             bit.clear();
10            ... n.A. ...
11        }
12    }
13
14    public static void Main(String[] args) {
15        new Thread(new Worker()).start();
16        ...
17        new Thread(new Worker()).start();
18    }
19 }
```

# Aufgabe: Mutex – Nachteile+Auswege

- aktives Warten
  - effiziente Implementierung von BS-Semaphoren
  - mit Scheduling integriert
- „P-V“-Reihung manuell+fehlerträchtig
  - Java-Sprachkonstrukt synchronized
  - Monitor

# Und nun...

## 3 Aufgabe: Bounded Buffer

# Aufgabe: Bounded Buffer – Aufgabe

- Implementieren Sie eine Klasse für PV-Chunk Semaphore
- Nutzen Sie PV-Chunk Semaphore für das Koordinierungsproblem „Bounded Buffers“
- was müssen Sie koordinieren?
- welche Semaphore benötigen Sie dazu?
- Unterschiede zu den Leser/Schreiber-Problemen?

# Aufgabe: Bounded Buffer – Quellcode PV-Chunk

```
1 public class PVChunk {
2
3     private int value;
4     public PVChunk(int max) { value=max; }
5
6     public synchronized void p(int amount) {
7         while(value<amount) {
8             try{ wait(); }
9             catch(InterruptedException e) {}
10        }
11        value-=amount;
12    }
13
14    public synchronized void v(int amount) {
15        value+=amount;
16        notifyAll();
17    }
18 }
```

# Aufgabe: Bounded Buffer – Lösungsskizze

- PV-Chunk werden nur mit  $P(1)$ ,  $V(1)$  verwendet!
  - als binäre Semaphore
- Was muss koordiniert werden?
  - Gegenseitiger Ausschluss **aller** bei Zugriff
    - Mutex als  $PV(1)$
  - Blockieren beim Lesen von leerem Puffer
    - $PV(0)$
  - Blockieren beim Schreiben in vollen Puffer
    - $PV(N)$
- Unterschied zu Leser/Schreiber-Problemen?
  - bei BB wird Datum beim Lesen aus Puffer entfernt!

# Aufgabe: Bounded Buffer – Quellcode

```
1 public class BoundedBuffers {
2
3     private char[] buf;
4     int first=0, last=0;
5     PVChunk full=null, empty=new PVChunk(0), mutex=newPVChunk(1);
6
7     public BoundedBuffers(int length) {
8         buf = new char[length];
9         full = new PVChunk(length);
10    }
11
12    public void put(char c) {
13        full.p(1);
14        mutex.p(1); ++last %= buf.length;
15                buf[last] = c; mutex.v(1);
16        empty.v(1);
17    }
18
19    public char get() {
20        empty.p(1);
21        mutex.p(1); char c = buf[first];
22                ++first %= buf.length; mutex.v(1);
23        full.v(1); return c;
24    }
25 }
```

# Und nun...

## 4 Studie: Arithmetik in Java

# Studie: Arithmetik in Java

- Problem: nicht alle Operationen sind atomar
  - abhängig vom Datentyp
- Long-Arithmetik ist i.A. nicht unterbrechungsfrei
- Beispiel
  - kontinuierliche Addition von `0x0000000100000001L`
  - intern realisiert über zwei Teiladditionen
    - Hi- und Lo-Word werden getrennt addiert
  - Inkonsistenzen ohne Koordinierung (synchronized)

## Studie: Arithmetik in Java – Quellcode

```

1  public class Coord implements Runnable {
2
3  static long n=0;
4  long nn=0, loop=0;
5  ...
6  public void run() {
7      while(true) {
8          nn = (n+=0x00000000100000001L);
9          int lo = (int) ( nn & 0xffffffffL );
10         int hi = (int) ( (nn >> 32) & 0xffffffffL );
11         if (lo!=hi) {
12             System.out.println(...);
13             n=((long)hi)<<32L | (long) hi; // restore (uncoord ;--)
14         }
15         loop++;
16         if (loop%10000000L==0) System.out.println(label);
17     }
18 }
19 public static void main(String[] args) {
20     new Thread(new Coord("t1")).start();
21     new Thread(new Coord("t2")).start();
22 }
23 }

```

# Studie: Arithmetik in Java – Ausgabe (Auszug)

```
1 t1
2 t2
3 t1
4 t2
5 t2 hi=2782ea4 not lo=279df50 distance=110764
6 t2 hi=28026b1 not lo=2837a59 distance=218024
7 t2 hi=2affe95 not lo=2b1a3a6 distance=107793
8 t1
9 t2
10 t1
11 t2
12 t2 hi=50bc44c not lo=50d6bee distance=108450
13 t1
14 t2
15 t1
16 t2
17 t1 hi=7c4d98a not lo=7c67109 distance=104319
18 t1
19 t2 hi=8176d64 not lo=81ac52b distance=219079
20 t2
21 t1 hi=8e2cae6 not lo=8e5fad5 distance=208879
22 t1
23 t2
```